REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		FINAL/01 FEB 93 TO 28 FEB 95

4. TITLE AND SUBTITLE
TOWARDS A FORMALISM FOR PROGRAM GENERATION 1995

5. FUNDING NUMBERS
2304/FS
F49620-93-1-0152

6. AUTHOR(S)
DANIEL E. COOKE

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TEXAS EL PASO
EL PASO, TEXAS 79968-0518

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR- 95-0416

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
110 DUNCAN AVE, SUTE B115
BOLLING AFB DC 20332-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
F49620-93-1-0152

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

BagL is a formal, general purpose language which provides for database, event-driven, and scientific computing in a uniform level of representation. In BagL a problem solver is not required to provide the algorithmic detail in a problem solution. Instead the problem solver describes the solution directly by specifying, via a metastructure, the data structures which will hold results useful in solving the problem. A metastructure is a very general abstraction configurable into any imaginable data structure, where a data structure is viewed as a database; a CRT screen or report layout; or a classical data structure such as a stack, a queue, ect. BagL provides a platform to specify the contents and the form of a data structure. BagL is an imparovement over current programming paradigms in that it provides for a natural platform to exploit data flow parallelisms and because it eliminates much of the technical complexity of problem solving including: decisions about data structures; decisions about how control structures are to interact with data structures; decisions about how one converts an external structure to an internal structure and vice versa; input-output decisions, in general; decisions about aprallelisms; decisions about control structures in general. BagL is a small language (i.e., there are few language constructs) and it is not domain dependent.

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR(SAME AS REPORT)

TOWARDS A FORMALISM FOR PROGRAM GENERATION 1995 - FINAL REPORT

Prepared by
Daniel E. Cooke

Department of Computer Science
University of Texas El Paso
El Paso, Texas 79968-0518

April 1995

Contract F49620-93-1-0152

Prepared for

AFOSR
Air Force Office of Scientific Research
Mathematical and Information Sciences
Air Force Office of Scientific Research
Bolling Air Force Base
Washington, D.C. 20332 6448

TOWARDS A FORMALISM FOR PROGRAM GENERATION 1995

Prepared by
Daniel E. Cooke

Department of Computer Science
University of Texas El Paso
El Paso, Texas 79968-0518

April 1995

Contract F49620-93-1-0152

Prepared for

AFOSR
Air Force Office of Scientific Research
Mathematical and Information Sciences
Air Force Office of Scientific Research
Bolling Air Force Base
Washington, D.C. 20332 6448

NOTICES

This final report is a presentation of the findings from research funded under F49620-93-1-0152.

ABSTRACT

BagL is a formal, general purpose language which provides for database, event-driven, and scientific computing in a uniform level of representation. In BagL a problem solver is not required to provide the algorithmic detail in a problem solution. Instead the problem solver describes the solution directly by specifying, via a metastructure, the data structures which will hold results useful in solving the problem. A metastructure is a very general abstraction configurable into any imaginable data structure, where a data structure is viewed as a database; a CRT screen or report layout; or a classical data structure such as a stack, a queue, etc. BagL provides a platform to specify the contents and the form of a data structure.

BagL is an improvement over current programming paradigms in that it provides for a natural platform to exploit data flow parallelisms and because it eliminates much of the technical complexity of problem solving including: decisions about data structures; decisions about how control structures are to interact with data structures; decisions about how one converts an external structure to an internal structure and vice versa; input-output decisions, in general; decisions about parallelisms; decisions about control structures in general. BagL is a small language (i.e., there are few language constructs) and it is not domain dependent.

TABLE OF CONTENTS

FOREWORD

This is the final report associated with AFOSR contract, F49620-93-1-0152.

PREFACE AND ACKNOWLEDGEMENTS

Research sponsored by the Air Force Office of Scientific Research (AFSC), under contract F49620-93-1-0152. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

OBJECTIVES.

In proposing the research effort which serves as the subject of this report, we set out to complete or make substantial progress toward each of the following research objectives:

(1). Complete a BagL interpreter based upon the current syntax and semantics;
(2). Enhance the semantics of BagL to abstract out quantifier information;
(3). Show expressiveness (via LISP approach);
(4). Extend the semantics of BagL to provide facilities for realtime specification and concurrency;
(5). Establish a logical semantic for BagL to provide for the statement of constraints;
(6). Provide automatic facilities to help maintain BagL software by employing results from the study of nonmonotonic logic; and
(7). Develop a prototype BagL environment including a visual interface.

In the next section, the status of each of these efforts is discussed.

STATUS.

In the following subsections each of the research objectives of the referenced contract is discussed.

(1). Complete a BagL interpreter based upon the current syntax and semantics

The first version of a BagL interpreter was completed in the summer of 1993 and reported in the 1993 yearly report [C94].

(2). Enhance the semantics of BagL to abstract out quantifier information

The BagL semantics were completed in the spring of 1993 and reported in the 1993 yearly report [C94]. These semantics abstracted out the quantifier information which was necessary in earlier versions of BagL.

(3). Show expressiveness

In March, 1995 we completed a proof that BagL is equivalent to the Universal Turing Machine. The basic parts of this proof are given in Appendix A.

(4). Extend the semantics of BagL to provide facilities for realtime specification and concurrency

The semantic extensions for BagL's concurrent and event driven capabilities exist in the semantics reported in the 1993 yearly report [C94]. The concurrent characteristics of BagL are given in Appendix B of this report.

(5). Establish a logical semantic for BagL to provide for the statement of constraints

A further extension to BagL, to impose integrity constraints, is accomplished and discussed in detail in the Ph.D. thesis of Ann Quiroz Gates [G94]. An overview of these concepts is given in Appendix C.

(6). Provide automatic facilities to help maintain BagL software by employing results from the study of nonmonotonic logic

The reliability of a program is subject to **environmental** influences. Programs do not operate in a vacuum. They are expected to fit into some environment and operate reliably in that environment. Unfortunately, environments change and specifications which were true in the original environment may not be true in future environments. When the truth of a specification changes as a result of an environment, the corresponding program must evolve to conform to the new environment. Evolution, therefore, is achieved through adaptive maintenance.

Environmental changes are indicated when a system is presented information which is inconsistent with the program's knowledge. A program cannot respond correctly when environmental information contradicts specified information.

In [LC95] it is shown that many environmental changes result in an inconsistency in the software system's specification. Using the integrity constraints discussed in Appendix C, the shifts in knowledge which result in inconsistencies are easily detected in constraint functions which operate on a DB and its successor, DB' (e.g., when some information is true in DB and false in DB'). Detecting inconsistencies permits the system itself to alert programmers of the need for adaptive maintenance. Thus, through its features for integrity constraints, it is possible that BagL may provide additional support for maintenance and, as a result, further improve upon the reliability of systems developed in BagL.

(7). Develop a prototype BagL environment including a visual interface

A Visual Interface and prototype environment are discussed in detail in the Master's Thesis of Aida Gandara [AG94]. An overview of this effort is given in Appendix D.

PUBLICATIONS.

Further evidence of the success of our theoretical work can be observed in our recent publications. The following is a list of journal papers (appearing or accepted to appear during the project period) based on the AFOSR project at UTEP.

C.V. Ramamoorthy, Daniel E. Cooke, and Chitta Baral, "Maintaining the Truth of Specifications in Evolutionary Software," *International Journal of AIT,* Vol. 2, No. 1 (1993) pp. 15-31.

Daniel E. Cooke, "Possible Effects of the Next Generation Programming Language on the Software Process Model," *International Journal on Software Engineering and Knowledge Engineering,* Vol 3 No 3, (September, 1993) pp. 383-399.

Daniel E. Cooke, "An Introduction to the Issues of Computer Aided Software Engineering," in *The Impact of CASE Technology on Software Processes,* World Scientific Publishing, Singapore (1994) pp 1-12.

Daniel Cooke, Richard Duran, Ann Gates, and Vladik Kreinovich, "Geombinatoric Problems of Environmentally Safe Manufacturing and Linear Logic," Vol. 4 No. 2 (October, 1994) , *Geombinatorics,* pp. 36-47.

Daniel Cooke, Elif Demirörs, Onur Demirörs, Ann Gates, Bernd Krämer, Murat M. Tanik, "Languages for the Specification of Software," to appear in *Journal of Systems and Software.*

Daniel Cooke, "An Informal Introduction to a High Level Language with Application to Interval Mathematics," to appear in *Interval Computations.*

Luqi and Daniel E. Cooke, "Rapid Prototyping and a Model for Software Maintenance," to appear in *International Journal on Software Engineering and Knowledge Engineering.*

During the contract period, the following conference papers were accepted and/or published and presented:

Daniel E. Cooke, "Arithmetic Over Multisets Leading to a High Level Language," PD-Vol. 49, *Computer Applications and Design Abstraction,* ASME 1993, Houston, Texas (January, 1993) pp. 31-36.

Daniel E. Cooke, "A High Level Programming Language Based Upon Ordered Multisets," *Proceedings of IEEE Fifth International Conference on Software Engineering and Knowledge Engineering,* San Francisco, (June, 1993) pp. 117-124.

Daniel Cooke and Luqi, "Formal Support for Software Maintenance," *IEEE COMPSAC '93,* Phoenix, AZ, (November, 1993) pp. 402-407.

Daniel E. Cooke, "A High Level Language For Engineering Applications," PD-Vol. 59, *Software Systems Engineering,* ASME 1994, New Orleans, Louisiana (January, 1994) pp. 323-329.

Daniel E. Cooke, Richard Duran, Ann Gates, and Vladik Keinovich, "Bag Languages, Concurrency, Horn Logic, and Linear Logic," in *Proceedings of IEEE Sixth International Conference on Software Engineering and Knowledge Engineering,* Riga, Latvia, (June, 1994) pp. 289-297.

Daniel E. Cooke, "A Formal Model of Problem Solving and its Impact on Software Development," *1994 Monterey Workshop on Formal Methods Proceedings,* Monterey, CA, (September, 1994), pp. 63-67.

Daniel E. Cooke, "Preliminary Thoughts Concerning the Interphase Activity of Requirement Migration," to appear in *Proceedings of IEEE Seventh International Conference on Software Engineering and Knowledge Engineering.*

Ann Q. Gates and Daniel E. Cooke, "The Use of Integrity Constraints in Software Engineering" to appear in *Proceedings of IEEE Seventh International Conference on Software Engineering and Knowledge Engineering.*

PERSONNEL.

The AFOSR research effort at the University of Texas at El Paso (UTEP) is staffed by three people. Dr. Daniel Cooke receives three month's salary each year as project director and principal researcher. Ms. Ann Gates and Mr. Richard Duran both received their twelve month salaries from the project.

Mr. Duran is completing a Master of Science in Computer Science in 1995 and Dr. Gates completed her Ph.D. in Computer Science at New Mexico State University in December, 1994. The research component of their respective degrees is based on the research they conducted for the project.

In August, 1993 Bassam Chokr (Previously funded by the AFOSR contract) received a Master of Science Degree in Computer Science. His thesis is entitled, *A Data Structure for BagL.*

INTERACTIONS.

During the contract period we have had close interactions with a number of people. All of these interactions have had a technical focus on the AFOSR funded research:

Alfs Berztiss, University of Pittsburgh

Jacob Schwartz, Courant Institute, NYU

Valdis Berzins, Naval Postgraduate School

Luqi, Naval Postgraduate School

C.V. Ramamoorthy, U.C. Berkeley

Murat Tanik, Southern Methodist University

S.K. Chang, University of Pittsburgh

Evidence of further professional interactions Cooke had during the contract period, follows.

Editorships:

Associate Editor of the *Journal for Software Engineering and Knowledge Engineering.*
Book Review Editor of the *Journal for Software Engineering and Knowledge Engineering.*
Co-editor *COMPUTER APPLICATIONS AND DESIGN ABSTRACTION 1992 - ASME.*
Editor *COMPUTER APPLICATIONS AND DESIGN ABSTRACTION 1993 - ASME.*
Proceedings Editor *COMPSAC 1993.*
Editor *SOFTWARE SYSTEMS IN ENGINEERING 1994 - ASME.*
Editor *SOFTWARE SYSTEMS IN ENGINEERING 1995- ASME.*
Proceedings Editor *IEEE ISADS 1995.*

Program Committees (member):

Sixth International Conference on Software Engineering and Knowledge Engineering (S.K. Chang, Conference Chair).
Seventh International Conference on Software Engineering and Knowledge Engineering (S.K. Chang, Conference Chair).
Third IEEE Systems Integration Conference (Raymond Yeh and Peter Ng, Conference Co-Chair).
IEEE CAIA '93 (Peter Selfridge Conference Chair).
IEEE COMPSAC '93 (Joe Urban Chair).
IEEE COMPSAC '94 (C.V. Ramamoorthy Chair).
IEEE ISADS '95 (Joe Urban Chair).

Program Committees (officer):

Symposium Chair *ASME Computer Applications and Design Abstraction '93.*
Symposium Chair *ASME Computer Applications and Design Abstraction '94.*
Chair *ASME Computer Applications and Design Abstraction '95.*
Chair *ASME Computer Applications and Design Abstraction '96.*
Chair of Workshop on Software Automation for Systems Integration Conference, 1992.
Chair of Workshop on Software Automation for International Conference on Software Engineering and Knowledge Engineering, 1993.
Chair of Workshop on Software Automation for International Conference on Software Engineering and Knowledge Engineering, 1994.

Demonstration Chair, SEKE '95.

Consulting and Technology Transfer:

During the contract period a number of promising application areas of BagL have been identified, including the Database area, C3I simulations, processing satellite telemetry data, etc. Below, these areas are discussed. Industrial interest in BagL has come from two companies, Knowledge Based Systems, Inc. and Research Analysis and Maintenance, Inc.

1. Multilevel Secure Databases.

Mr. Jim Reed of the Data Analysis Center in Alexandria, Virginia (703 960-1000) is interested in BagL for the purpose of developing and modeling multilevel secure databases.

2. C3I Battlefield Management Specification and Prototyping

It is believe that BagL possesses the features necessary to become an excellent tool for specifying and modeling C3I systems. Mr. Sam Dinitto, Division Chief at RADC (315 330-3011) supports the claim that BagL holds promise as a language for the specification of Battlefield Management Software. In particular, Mr Dinitto claims that BagL is applicable to the specification of C3I Systems in the Systems Definition Technology Division of Rome Labs.

3. Rapid Prototyping and Automatic Programming of Naval Applications.

Mr. William McCoy (703 663-8367) of the Naval Surface Warfare Center is interested in BagL for the prototyping and specification of Naval Warfare systems such as the AEGIS System. Ultimately a BagL compiler is to be developed which will compile BagL specifications to "C" or Ada programs. The NSWC is also interested in a prototyping tool that is capable of automatically producing programs. Since BagL is a complete, executable language, it is possible to build such an automatic program generator.

Professor Alfs Berztiss (412 624-8401) of the University of Pittsburgh believes that BagL is an excellent language for specification.

4. Rapid Development of Programs to Process Satellite Telemetry Data.

Officials of NASA Ames research center have recently begun the development of a new satellite called MEDSAT. This satellite is to be used to track the spread of disease carried by insect populations. BagL is proposed to be the language used for processing the resulting telemetry data. The BagL research project receives some modest funding from NASA Ames.

5. Education of Computer Science Students.

Valdis Berzins, currently a professor at the Naval Postgraduate School (408 656-2461), has had a long career in language development. This career spans across the institutions MIT, University of Minnesota, and the Naval Postgraduate School. Professor Berzins recently learned BagL and claims that all Computer Science students should learn this language. He supports his comments by stating that BagL abstracts out everything unnecessary in problem solution; that its computational model is as simple as it can possibly be; there is nothing there that does not have to be there. Furthermore, it can serve as a tool to introduce database and concurrency concepts.

INTELLECTUAL PROPERTY ISSUES.

No applications for patents or copyrights have been made.

ADDITIONAL STATEMENTS.

Additional goals which exceed the objectives of the research effort include the development of a second prototype interpreter for BagL in 1994, the initial steps to develop a proof system for BagL programs, and the identification of constructs for processing nonscalar structures. Appendices E and F summarize the two latter results.

APPENDIX A - BAGL EXPRESSIVENESS

Foundation

The foundation of this work rests upon the concepts of the Turing machine and Church's Thesis. Alan Turing's conceptual automata is able to perform any computation known to man. Attempts to improve it through modification have not increased its performance and it remains the best definition of an algorithm that we have today. Alonzo Church's thesis that no computational procedure will be considered an algorithm unless it can be presented as a Turing machine has never been refuted. Every other computational procedure developed has been shown equivalent to a Turing machine. The combination of these two concepts provides us with a basis for universality, the ability to represent any algorithm. The ability to simulate any arbitrary Turing machine in the BagL programming language demonstrates that any algorithm can also be represented in BagL, thus establishing the universality of BagL

The BagL Turing Machine

We began by describing and defining the concept of a Turing machine as a mathematical definition of an algorithm. It followed that any algorithm can be represented by a Turing machine. After defining Turing machine configurations and computations, we illustrated three computational examples. We then provided all the pieces necessary to simulate any arbitrary Turing machine from functions and data structures in the BagL programming language. The ordered bag data structure simulated the Turing machine tape and head. BagL built-in functions provided the "move right," "move left" and "replace" functions for the Turing machine's finite control. Refining these three functions to prevent "hanging" and alleviate minor problems with bag markers gave us three functions that serve as the BagL Turing machine's finite control under any conditions.

We next created complex BagL functions to simulate the individual computational steps on a given Turing machine. We thus described how to simulate all components of any Turing machine in BagL, thereby enabling us to simulate any arbitrary Turing machine in BagL. We concluded by illustrating the same three examples computed earlier on a Turing machine, this time using the BagL Turing machine. The results were identical, both in number of computational steps and in configurations. The definitions and examples indicate that, using eventive constructs, BagL can replicate any arbitrary Turing machine without the use of recursion.

APPENDIX B EVENTIVE CONSTRUCT

The body of a *BagL* function is based upon Djikstra's guarded commands [D75]. A *BagL* program consists of a set of *BagL* functions which are applied to a *BagL* **Universe**, *U*. The **Universe** is where the user may pair *BagL* variable names with ***bags*** of values via a text editor.

Like **Linda** [G85] when a *BagL* program executes it modifies the **Universe** to which it is applied. If each variable appearing in a function's *Domain* is paired with a ***bag*** in *U* (and if relations are present in the Domain, each relation evaluates to true with respect to *U*), the function is enabled for execution and is termed an **outer function**. Obviously, several functions may be thus enabled. These functions may execute concurrently. When an outer function executes, it consumes all variables in its domain.

When an outer function completes execution, the result(s) of the function is(are) paired with the function's range variable(s) and added (or produced) in the **Universe**. It is via the **Universe** that a user provides inputs and obtains outputs. There are no explicit constructs for I/O in *BagL* . *BagL* **inner functions** are invoked by some function that is already executing. A *BagL* inner function *f* receives its input ***bags*** from the invoking function *g* and returns its result (always a ***bag***) to *g*.

The interaction between a *BagL* function and its **Universe** in terms of production and consumption provides for the ***eventive processing of a nonscalar structure.*** In eventive processing, one processes a nonscalar structure whose elements are not available all at once. The arrival of an element is an event to which a function must respond.[1]

Example of the Execution of a *BagL* Program.

Consider the following set of functions to compute the average deviation of a set of numbers. Assume an initial **Universe**:

$$U = \{<x,[[33],[88],[66],[44]]>\}$$

and a *BagL* program:

$$\Pi = \{\ ave(\ Domain(x),Range(x,ave)) = [\ x,\ /([+([x]),\mathbf{size}([x])])\ ]$$
$$dev(Domain(x,ave),Range(x,ave,dev)) =$$

---

[1] We are currently investigating the use of a fairness operator [D89] to address the nondeterminancy in the event-driven semantic of *SequenceL*, wherein two consuming outer functions have a non-null intersection of domain/range variables.

*[x, ave, /([+([abs([-([x,ave])])]),**size**([x]))]}*

The only function eligible for execution is *ave*. When it begins execution it consumes its domain so that *U={}*. A trace of the execution of *ave* follows. In each line, the term to be evaluated next is underlined. The nature of the evaluation appears as an annotation in the intervening lines.

*[x, /([+([x]),**size**([x]))]* »

Instantiate *x* based upon *U*

*[[[33],[88],[66],[44]], /([+([[33],[88],[66],[44]]),**size**([[33],[88],[66],[44]]))]* »

Evaluate the *addition* operation

*[[[33],[88],[66],[44]], /([[231],**size**([[33],[88],[66],[44]]))]* »

Evaluate the *size* operation

[[[33],[88],[66],[44]], /([[231],[4]])] »

Evaluate the *division* operation

[[[33],[88],[66],[44]], [57.75]]

The completion of this outer function results in the **Universe**:

U = {<x,[[33],[88],[66],[44]]>, <ave,[57.75]>}

Now, function *dev* is eligible for execution. An annotated trace of its execution follows:

*[x, ave, /([+([abs([-([x,ave])])]),**size**([x]))]* »

Instantiate *x* based upon *U*

[[[33],[88],[66],[44]], [57.75], /([+([abs([-([[[33],[88],[66],[44]],[57.75]])])]),
size([[33],[88],[66],[44]]))] »»

Evaluate the *subtraction, abs,* and *size* operations

[[[33],[88],[66],[44]], [57.75], /([+([[24.75],[30.25],[8.25],[13.75]]),[4]])] »

Evaluate the *addition* operation

[[[33],[88],[66],[44]], [57.75], /([[77],[4]])] »»

Evaluate the *division* operation

[[[33],[88],[66],[44]], [57.75], [19.25]]

The completion of this outer function results in the **Universe:**

U = {<x,[[33],[88],[66],[44]]>, <ave,[57.75]>, <dev,<19.25]>}

Notice that the result of *ave* is required for the execution of *dev*. Furthermore, notice that *dev* could invoke *ave* to obtain the value needed, making *ave* an inner function:

*∏ = { ave(Domain(x),Range()) = [/([+([x]),**size**([x])])]*
dev(Domain(x),Range(x,ave,dev)) =
*[x, /([+([abs([-([x,ave([x])])])]),**size**([x])])]}*

which is evaluated according to the following annotated trace:

*[x, /([+([abs([-([x,ave([x])])])]),**size**([x])])]}* »
Instantiate *x* based upon *U*
[[[33],[88],[66],[44]] , /([+([abs([-([[[33],[88],[66],[44]],
*ave([[33],[88],[66],[44]])])])]),**size**([[33],[88],[66],[44]])])]* »»
Evaluate the *ave* user-defined function
[[[33],[88],[66],[44]] , /([+([abs([-([[[33],[88],[66],[44]],
*[57.75]])])]),**size**([[33],[88],[66],[44]])])]* »»
Evaluate the *subtraction, abs,* and *size* operations
[[[33],[88],[66],[44]], /([+([[24.75],[30.25],[8.25],[13.75]]),[4]])] »
Evaluate the *addition* operation
[[[33],[88],[66],[44]], /([[77],[4]])] »»
Evaluate the *division* operation
[[[33],[88],[66],[44]], [19.25]]

In all previous examples, the eventive construct is employed on *all* elements of the domain structure. It is also possible to selectively execute an outer function on *some* of the elements, rather than on all. In these cases, one employs a condition on the domain variables:

cool(Domain(>(temp,thermostat),=([ac_on,[false]])),Range(ac_on,temp,thermostat)) =
[[true],temp,thermostat]
steady(Domain(<=(temp,thermostat),=([ac_on,[true]])),Range(ac_on,temp,thermostat)) =
[[false],temp,thermostat]

In this example, not only must values exist for domain variables *temp* and *thermostat,* the domain conditions on these variables must be satisfied. Notice that current *temp* and *thermostat* values are reset since they appear as Range variables.

Based upon the *BagL* ***data dependency execution strategy,*** it is clear that many functions may be eligible for execution at any given time. Important characteristics of the concurrent behavior of *BagL* outer functions can be ascertained by the Domain/Range sets of variables given in the function definitions. *BagL* supports different modes of operation. The actual mode is dependent upon whether domain and/or range variables are shared among outer functions. Assume that *f(DOM)* and *f(RAN)* are the complete sets of variables referenced in *f's* domain and range definitions, respectively. Assuming that both *fi* and *fj* are outer functions, the following definitions follow logically:

Def 1. Competing Functions. Functions *f*i and fj are competing iff
$fi(DOM) \cap fj(DOM) \neq \varnothing$.

Def 2. Interfering Functions. Functions *f*i and fj are interfering iff
$fi(RAN) \cap fj(RAN) \neq \varnothing$.

Def 3. Cooperating Functions. Functions *f*i and fj are cooperating iff
fi and fj are not Competing or Interfering and
$fi(DOM) \cap fj(RAN) \neq \varnothing \vee fj(DOM) \cap fi(RAN) \neq \varnothing$.

Def 4. Autonomous Functions. Functions *f*i and fj are autonomous iff they are not competing, cooperating, or interfering.

Outer functions that satisfy either definition, 1 or 2 (i.e., Competing or interfering functions), result in concurrent, nondeterministic behavior. Functions that satisfy only the Cooperating functions definition are sequential, and those satisfying definition 4 result in concurrent, deterministic behavior.

APPENDIX C INTEGRITY CONSTRAINTS IN BAGL

BagL IS extended to impose database integrity constraints through the use of the ***eventive*** construct of BagL which is defined in Appendix B. The extension does not add to the knowledge required of the BagL programmer. Functions which impose integrity constraints behave in a manner consistent with the behavior of BagL functions which compute results. Consider a simple example. Assume a database exists, like that presented in figure C.1, view 1.

A BagL function to compute a ten percent raise for all employees, as applied to the Employee Database, would be:

Function Raise(Dom(Salary), Ran(Salary)) **is** [Salary * 1.1]

One type of BagL function, called an **outer function**, is initiated for execution based upon the availability of data in the database. In particular, the domain variables of the function (e.g., Dom(Salary)) must be paired with values in the database to which the function is applied. (The data dependent execution strategy provides the BagL eventive construct.) When the function begins execution, the domain variables are consumed from the database. Assuming **Raise** is an outer function, the initiation of its execution results in the change to the database as it is depicted in view 2 of figure C.1.

View 1

	Name	Salary
Employee Database	bob	50,000
	mary	55,000
	sue	90,000

View 2

Name
bob
mary
sue

View 3

Name	Salary
bob	55,000
mary	60,500
sue	99,000

Figure C.1. Modifying the Employee Database.

When an outer function completes execution, its results are paired with the range variable(s) and placed in the database. For example, when function **Raise** completes execution, its result is produced in the database and paired with the name **Salary**. See figure C.1, view 3.

In order to impose constraints on a database, it is typically incumbent upon the programmer to distribute appropriate guards into all functions which update a constrained field. For example, suppose there is a cap placed on salaries in the Employee Database above, limiting salaries to be no greater than $100,000. To impose this constraint, the programmer in a typical language must place an appropriate guard on any statement

updating the salary attribute. Furthermore, the programmer must be concerned about **indirect** updates to the constrained attribute. For example, when a new employee's information is added as a tuple to the Employee Database, the programmer must recall that there is a constraint on **Salary** and furthermore, the programmer must recall that **Salary** is an attribute of the Employee Database.

The problem with relying on the programmer to impose constraints is that the programmer has to keep track of the constraints and know when it is appropriate to impose them. When a constraint changes, a system maintainer must remember all of the functions which are affected by the change and make the appropriate modifications.

One can envision BagL functions which initiate execution upon availability of databases rather than upon domain variables. We are making simple alterations to the syntax and semantics of BagL so that constraint functions will behave and be written in a manner consistent to BagL computational functions:

```
Constraint(Dom(DB,DB'),Ran(DB)) is
    [     [DB']  when <=(Salary',100,000)
          [DB]   otherwise      ]
```

This function executes whenever the named Database and its successor (denoted by a prime) is available. Whenever the named database is updated (i.e., when both DB and DB' become available), the function, **Constraint**, returns the new database if the Salary constraint is met. Otherwise, the old database is returned. There is no need for the programmer to place guards throughout a program or set of programs which affect, either directly or indirectly, the constrained attribute. The constraint is stated one time and is imposed on **any** update, whether direct or indirect. If, at a later time, the constraint changes, only one software update is necessary. By removing the responsibility for imposing constraints from the programmer, BagL provides a foundation for the production of more reliable software.

APPENDIX D VISUAL LANGUAGE

The representation of a problem solution in any language, regardless of the abstraction level of the language, is complicated by two needs:

(1). the need to present nested operations and
(2). the need to select operands upon which the operations are to apply.

Nested program structures are a major source of confusion when attempting to read or write a program. In procedural programs this source of complexity is made worse by the fact that there are many different types of control structures that can be nested, including procedures, functions, iterative statements, and if-then-else structures. Programmers must understand the interaction between nested statements of differing constructs (e.g., loops inside if-then-else, etc.) and then they must understand the interaction of the non-control statements inside the nested structures.

BagL presents an improvement over the procedural languages in that **only** non-control statements are nested in BagL (Control statements, other than guarded commands, do not exist in BagL). Therefore, the programmer need only understand the interaction of nested terms and relations in BagL -- nothing else can be nested. However, BagL still suffers from some of the confusion which results from nested structures. Consider the irregular BagL function to compute the square of a matrix:

function square(dom(t), ran(sq)) **is** [+([*([t(i, (*)),t((*),j)])])]

This function is indeed daunting. It states that for each **i** and **j** associated with table **t** multiply row **i** by the corresponding elements of column **j**. This operation results in a bag of products to which the **+** sign is distributed, resulting in the singleton bag corresponding to the **i,j**th position of the range bag, **sq**.

With the **square** function, one observes the difficulty of representing nested structures; we call this problem the **containment** problem. What is inside what? The typical approach to dealing with this problem is to use a directed graph to represent the flow of the equation. [B93] See figure D.1.

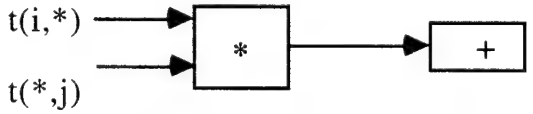

Figure D.1. Nested Structures.

While an improvement over the textual definition, there may be better ways to indicate the nesting of BagL terms. Using our method, the containment of an operation is self-evident. Consider again the BagL function to square a matrix, but this time the presentation will benefit from an improved visual approach:

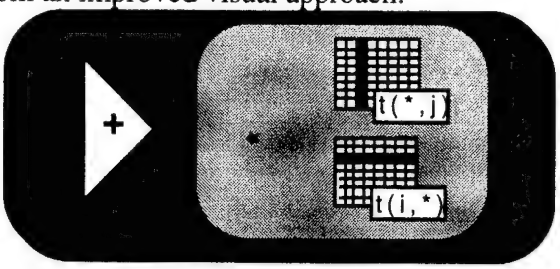

Figure D.2. Containment.

As one can observe, the containment of an operator is self-evident when compared to the textual function, above. BagL's abstraction actually facilitates the development of a visual interface. The research into visual languages and interfaces is typically based upon current paradigms of programming. The languages serving as a foundation for this work have complicated computational models which tend to muddy the waters in the search for interface improvement. As stated previously, BagL has the simplest possible computational model. There are no unnecessary complications in the model.

Similarly, the BagL model suggests basic operations associated with the selection of operands for computation. There seem to be basic selection operations which may also have revealing visual definitions. The two fundamental ways to select data are (1). based upon value and (2). based upon position. The darkened row and column of tables **t** in figure D.2 suggest a scheme to depict the selection of operands based upon position. With a better representation scheme, BagL should provide a view of problem solution that is easy to read, write, and comprehend. With better comprehension should come increased effectiveness.

APPENDIX E A PROOF SYSTEM FOR BAGL PROGRAMS

In this appendix we introduce the axiomatic definitions of BagL outer functions. We will consider only outer functions with no cooperation, i.e., $\sim(\exists f1, f2)(f1 \neq f2 \ \& \ Domain(f1) \cap Range(f2) \neq \{\})$, where fi is a BagL function. We give all definitions necessary to relate the example BagL function with a corresponding formal requirement.

The idea in the requirements phase is that one is stating his/her intentions with respect to the affect a BagL function is to have on its universe. In the following, the formula R&V&D is derived from a **formal** requirement and states the desired result of a function. In the formula R&V&D, R is the condition desired, D is a formula defining the two sets DOMAIN and RANGE, and V is a formula which states membership information for the Universe, U.

Ax1: $wp("E", R) = R^{x}_{E}$ & {x}=RANGE

where "R^{x}_{E} denotes a copy of the predicate defining R in which each occurrence of the variable *x* is replaced by [the meaning of a BagL_Term], (E)" [D75].

Ax2: $wp("E_i \ when \ B_i", R) = BB$

where $BB = (\exists i : 1 \leq i \leq n : B_i)$ &
$(\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(E_i, R))$

See [D75] for the theorems related to BB. In Ax3, assume *Dom* and *Ran* are the sets of domain and range variables given in the BagL function *f*'s definition:

Ax3: $wp("f(Dom,Ran)= [E_i \ when \ B_i \ ]", R\&V\&D) =$
$(wp(E_i \ when \ B_i, R)) \ \&$
$(\forall x)(x \in DOMAIN \Rightarrow wp(x,V)) \ \&$
(Dom=DOMAIN & Ran=RANGE)

Our goal with Ax2 is to make BB=TRUE. In doing so, the hope is that the first and third outer conjuncts of Ax3 will be true, leaving the second conjunct as the sole precondition to the function's execution. The second conjunct of Ax3 effectively gives the semantic of the function: that the elements of the DOMAIN are consumed in the universe, leading to the production of the RANGE element.

An Example.

Our goal in the proof is to show that a function of BagL satisfies the postcondition *R&V&D* which is obtained from a SPEC [BL91] requirement:

CONCEPT max(x,y:integer) VALUE (m:integer)
WHERE (m=x OR m=y) & m≥x & m≥y

Becomes,

R&V&D, where
R=(m=x OR m=y) & m≥x & m≥y --from the WHERE clause
V= U ⊇ {<m,_>} --from the VALUE clause[1]
D= DOMAIN={x,y} & RANGE={m}
--- from the arguments of CONCEPT max and the VALUE clause

We begin the derivation with Ax1 because in order to obtain a function we must first obtain its set of guarded commands, and in order to obtain a set of guarded commands, we must first obtain the individual commands -- the only commands possible in BagL are the BagL_Terms "E". Given R, we first let "E" be "x":

By Ax1 and through simplification:

wp("x",R) = ((m=x OR m=y) & m≥x & m≥y)$^{m}_{x}$ & {m}={m}
= ((x=x OR x=y) & x≥x & x≥y) & TRUE
= x≥y

By Ax2 (the second conjunct of BB) we can take the wp("x",R) as a guard:
(x when x≥y)

However, the first conjunct of BB, namely ($\exists i : 1 \leq i \leq n : B_i$), is not satisfied, i.e., BB≠TRUE. To make BB=TRUE we repeat the sequence above for the other alternative which will guarantee there will always be some B_i=TRUE:

By Ax1 and through simplification:

wp("y",R) = ((m=x OR m=y) & m≥x & m≥y)$^{m}_{y}$ & {m}={m}
= ((y=x OR y=y) & y≥x & y≥y) & TRUE
= y≥x

By Ax2 (the second conjunct of BB) we take the wp("y",R) as a guard:
(y when y≥x)

---

[1] Assume that underscore (i.e., "_") can match anything as an anonymous variable.

Leading to the guarded set

(x when x≥y else
y when y≥x)

which satisfies Ax2 (i.e., BB=TRUE), leading to the application of Ax3:

wp(*"f([x,y],[m])=(y when y≥x else x when x≥y)"* ,R&V&D)

= TRUE & (∀x)(x ∈ DOMAIN => wp(x,V)) &
(Dom=DOMAIN & Ran=RANGE) [By Ax2]

= TRUE & (wp(x,V) & wp(y,V)) &
({x,y}={x,y} & {m}={m}) [By Substitution]

= (wp(x,U ⊇ {<m,_>}) & wp(y,U ⊇ {<m,_>})) & TRUE
[By Simplification]

= U ⊇ {<x,_>} & U ⊇ {<y,_>}
[By Ax1 and simplification]

Now, the function, its precondition, and its postcondition (which was given) are known. One can see, as a result of the foregoing, that the BagL function satisfies R&V&D which was obtained from the SPEC requirement:

(U ⊇ {<x,_>} & U ⊇ {<y,_>})
f([x,y],[m])=(y when y≥x else x when x≥y)
((m=x OR m=y) & m≥x & m≥y) &
U ⊇ {<m,_>} &
(DOMAIN={x,y} & RANGE={m})

APPENDIX F NONSCALAR CONSTRUCTS

From one perspective, the history of language design indicates that languages have evolved from algorithmic-oriented languages to data structure-oriented languages. (See Figure F.1.) Consider the move from FORTRAN to languages like ALGOL and Pascal. In solving problems in FORTRAN, one focuses on the production of an algorithm. In a language like Pascal, a problem solution is an algorithm that interacts appropriately with a data structure. A good example of the Pascal view is a program to convert a prefix arithmetic expression into an equivalent postfix expression. After constructing a binary tree which holds the prefix expression in an appropriate configuration (that is, a configuration where operators are parents of operands), one merely performs a postorder traversal of the tree to obtain the postfix expression. The data structure designed thus, plays a central role in the problem solution.

Object-oriented languages, which encapsulate data structures, take the notion of problem solving further away from algorithms and closer to data structure design. The intent is to view the product of software engineering to be a set of objects, rather than programs.

BagL has been developed in order to explore the possibility of a language based solely on high level constructs for describing data structures. In *BagL* , a problem solution is a data structure which holds results useful in solving some problem. This view of a data structure (henceforward called a *nonscalar*) is meant to include traditional data structures, databases, screen displays, reports, etc. The nonscalars of *BagL* are ***bags***.

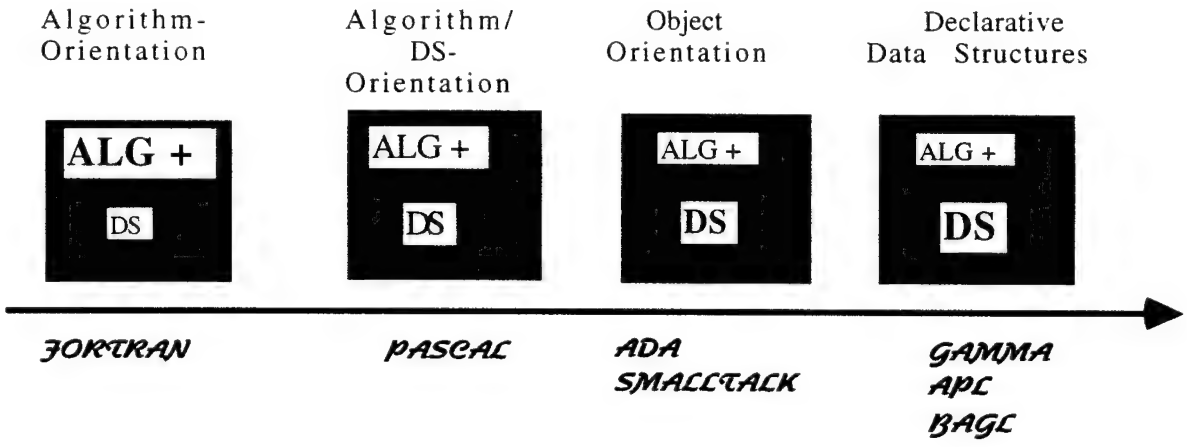

Figure F.1. Evolution of Languages.

Bags in *BagL* are collections of elements wherein each element may occur more than one time (as opposed to a mathematical *set*) and where each occurrence of an element possesses an ordinal position. A ***bag*** may be singleton (e.g., [99]), or nonsingleton

(e.g., [[1],[2],[3]] or [[[1],[2],[3]],[[10],[20],[30]]). Complex structures of ***bags*** containing ***bags*** can be described. Like the list of LISP [L60], the ***bag*** of *BagL* can be used to build any data structure.

When processing nonscalars, the central problem is the *selection* of elements to which the processing is to be applied. Sometimes it is desirable to apply an operator to *all* elements of a nonscalar. The ***regular*** form of processing (which is one of the nonscalar processing constructs of *BagL*) is one where an operation is to be applied in a uniform manner to all of the elements of the nonscalar. For example, when one computes the *sum* of a set of elements, the addition operator is being applied to *all* elements of the set (i.e., the addition operator is acting as an aggregate operator in this case). In other cases, one may wish to apply an operator to *some* of the elements of the nonscalar. The selection of elements may be based upon *value* or *position*. *BagL* possesses a construct for this ***irregular*** form of processing data.

In both the regular and irregular forms of processing nonscalars, one begins with a nonscalar and (typically) *reduces in the dimension* of the nonscalar (in the case of regular processing) or *reduces in cardinality* (in the case of irregular processing).[1] There are times, when it is necessary to *expand* a nonscalar, increasing the nonscalar in dimension and/or cardinality. *BagL* possesses a construct termed the ***generative*** construct which is used to expand nonscalars.

---

[1] A reducing construct, given an input D will produce a result, R where R is no larger (in dimension or cardinality) than D. E.G., Given a construct that reduces in cardinality: $D \rightarrow R$, the following relation holds after D is mapped to R: $|D| \geq |R|$

DOMAIN	SELECTION	RANGE	
		Reduction	Expansion
ALL ELEMENTS PRESENT	ALL	Dimension	Dimension
		Cardinality	Cardinality
	SOME	Dimension	Dimension
		Cardinality	Cardinality
SOME ELEMENTS PRESENT	ALL	Dimension	Dimension
		Cardinality	Cardinality
	SOME	Dimension	Dimension
		Cardinality	Cardinality

Figure F.2. A Summary of Nonscalar Constructs.

Sometimes, a nonscalar is being processed wherein all elements of the nonscalar are not present all at once. In this case, the arrival of an element is an event to which an operation must respond. *BagL* possesses an ***eventive*** construct. With (or without) the eventive construct there are features to select *all* or *some* of the elements in the nonscalar structure. See Figure F.2 for a summary of the constructs necessary to describe nonscalar processing.

In nonscalar processing, a nonscalar in some domain (i.e., the *DOMAIN* column in figure F.2) is selected to be the basis for a new nonscalar (i.e., the *RANGE* column in figure F.2). The input (or domain) nonscalar may be totally or only partially present. The ***eventive*** construct of *BagL* captures this latter notion. The first and third options in the second (i.e., the *SELECTION*) column indicate that given a nonscalar *X*, where *X* is to serve as the basis for a new nonscalar *Y*, all values of *X* are used to form *Y*. The ***regular*** construct satisfies this notion.

Alternatively, only selected values of *X* may serve to form *Y* (the second and fourth options in *SELECTION*). The ***irregular*** construct provides this facility of nonscalar processing. Finally, the result *Y* (or *RANGE*), when compared to its basis *X*, may be reduced in dimension or cardinality. The ***regular*** and ***irregular*** constructs reduce in dimension and cardinality, respectively. Alternatively, *Y* may be expanded in dimension or cardinality. The ***generative*** construct provides for the expansion of

nonscalars. The ***regular***, ***irregular***, ***generative***, and ***eventive*** constructs of *BagL* may be viewed as nonscalar primitives.

There are other declarative languages meant to process nonscalars. They typically excel in one form of nonscalar processing, but fall short in others. For example, one of the earliest languages for nonscalar processing, APL, excelled in the regular processing of nonscalars but often fell short in the other forms of nonscalar processing, most notably in the irregular processing of nonscalars.

REFERENCES

[AG94] Aida Gutierrez Gandara, "A Visual Interface for a Formal Speicifcation Language," Master's Thesis, University of Texas at El Paso, December, 1994.

[BL91] Berzins and Luqi, *Software Engineering with Abstraction,* Addison-Wesley, New York, 1991.

[B93] Alfs Berztiss, "The Query Language Vizla," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 5, No. 5, (October, 1993), pp. 813-825.

[C94] "Towards a Formalism for Program Generation," Daniel E. Cooke, for the Air Force Office of Scientific Research, #F49620-93-1-0152, February, 1994.

[D75] E. Djikstra, "Guarded Commands, Nondeterminancy and the Formal Derivation of Programs," *Communications of the ACM,* Vol. 18 No. 8, August, 1975, pp.453-457.

[D89] A. Deshpande and K. Kavi, "A Model for the Specification of Concurrent Processes," *Microcomputer Applications,* Vol. 8, No.3, March, 1989, pp. 95-102.

[G85] D.Gelernter, "Generative Communications in Linda", *ACM Transactions on Programming Languages and Systems*, **7**(1), pp. 80-112 (1985).

[G94] Ann Quiroz Gates, "Context Monitoring with Integrity Constraints," Ph.D. Thesis, New Mexico State University, 1994.

[LC95] Luqi and Daniel E. Cooke, "Rapid Prototyping and a Model for Software Maintenance," to appear in *International Journal on Software Engineering and Knowledge Engineering.*